# Combining B Tools for Multi-Process Systems Specification

Christian Attiogbé

# Combining B Tools for Multi-Process Systems Specification

**Christian Attiogbé**

Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière - B.P. 92208 - 44322 NANTES CEDEX 3

*— Formal Methods Integration, Reliable Systems —*


## RESEARCH REPORT

## N° 06.01

## Janvier 2006




LINA, Université de Nantes – 2, rue de la Houssinière – BP 92208 – 44322 NANTES CEDEX 3
Tél. : 02 51 12 58 00 – Fax. : 02 51 12 58 12 – http://www.sciences.univ-nantes.fr/lina/

Christian Attiogbé

Combining B Tools for Multi-Process Systems Specification

28 p.

Les rapports de recherche du Laboratoire d'Informatique de Nantes-Atlantique sont disponibles aux formats PostScript® et PDF® à l'URL :
http://www.sciences.univ-nantes.fr/lina/Vie/RR/rapports.html

*Research reports from the Laboratoire d'Informatique de Nantes-Atlantique are available in PostScript® and PDF® formats at the URL:*
http://www.sciences.univ-nantes.fr/lina/Vie/RR/rapports.html

# Combining B Tools for Multi-Process Systems Specification

Christian Attiogbé

Christian.Attiogbe@univ-nantes.fr

**Abstract**

We introduce a systematic method that combines process-oriented design and abstract systems to specify multi-process system using Event B. As far as specification is concerned, the proposed method guides the user to describe his/her system by considering both process-oriented view and event B style of specification.
With regard to mechanization, the method enforces the conjoined use of both theorem proving technique and model checking technique with the associated tools: AtelierB and ProB. This method makes it easy the specification in Event B of multi-process systems and it also fasters the correctness proof. Our proposal is illustrated with a case study: a multi-process system with the interaction between processes that deal with parts using common resources.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms: Multi-Process Specification, Event B, Theorem Proving, Model checking

Additional Key Words and Phrases: Multi-Process Specification, Event B, Theorem Proving, Model checking

# Contents

# 1 Introduction

The correct design and development of complex software systems is a major research topic as software systems are more and more present in everyday life and more and more complex according to their interactions with the environment. Process-oriented systems are well-suited as they handle the interaction features: concurrency and synchronization.
Several methods exist that enable the developer to specify and formally analyse her/his systems: finite state machines approaches, Petri nets, process algebra are the most popular (explicit) *state transition system* approaches. They are well researched, they have graphical notations and they are equipped with tools. The latter are often based on *model checking techniques* that explore the state space covered by the studied systems.
However the complexity of state transition systems increases quickly for real life systems; this depends on: the importance of both data and control parts (state space explosion); the number of processes involved in the system; the constraints on the resources used by the processes (sharing, limited number). Indeed the transition systems become huge, they are not mentally manageable for human being and their analysis is time-consuming.

Besides, there are the emergence of formal development techniques that emphasize the correct construction of software systems using refinement techniques: the B method [1] ranges in this category. Further developments of the B method, known as Event B [2, 4, 3], enable one to build more generally discrete event systems. With the Event B approach the design of a system begins with an abstract specification. The latter is characterized by a *state space* described with predicates and a *list of events* that describe the behaviour (transitions caused by the events) of the system. The verification of properties and the refinement of abstract specifications into concrete (executable) codes are supported by proof of properties: *theorem proving techniques* are used. The state space explosion is not a limitation here and infinite systems may be dealt with. The study of a system is a top-down approach going from an abstract model to a concrete one via refinements and decompositions [3]. This approach is not always trivial even for small-size systems and it may be very tedious for large and complex systems. It is less intuitive than the state transition approaches.

The *motivation* of our work is the search of efficient *specification and design methods* that combine process-oriented approach with B method in order to provide more guidance to developer and to get mechanized techniques that scale well so as to be helpful for large interactive (possibly distributed) applications. An important part of today software applications, (Internet-)service oriented applications, process control systems are reactive (distributed) applications. They need to be safe and reliable but formal development techniques help in ensuring these properties.

The *contribution* of the current work is a systematic method (practical guidelines) to specify and verify multi-process systems by combining well-established formal techniques: process-oriented approach, Event B and model checking. We illustrate the proposed method by a case study with interacting processes. The case study reveals some interesting features of B specifications: for example, dealing with the dynamically variable number of the interacting processes.

The article is structured as follows. In the Section 2 we introduce the used methods. The Section 3 is devoted to the proposed design support. In the Section 5 we summarize the results. In the Section 6 we conclude the article and we introduce some perspectives.

# 2 Event B Method and Tools

## 2.1 Event B: an Overview

Within the Event B framework, asynchronous systems may be developed and structured using *abstract systems*. *Abstract systems* are the basic structures of the so-called *event-driven* B, and they replace the *abstract machines* which are the basic structures of the earlier *operation-driven* approach of the B method[1].
An abstract system describes a mathematical model of an asynchronous system behaviour[1]; it is mainly made of a *space state* description (constants, properties, variables and invariant) and several *event* descriptions. Abstract systems are comparable to Action Systems [7]; they describe a non-deterministic evolution of a system through

[1] A system behaviour is the set of its possible state transitions beginning from an initial state.

guarded actions. Dynamic constraints can be expressed within abstract systems to specify various liveness properties [4]. Abstract systems may be refined like abstract machines.

```
SYSTEM          S
SETS            CS, SS
VARIABLES       gv
INVARIANT       Inv
INITIALISATION  U
EVENTS
ee_1 ≙ /* an event */
   ANY bv WHERE
      P1(bv,gv) ∧ P2(gv)
   THEN
      GS(gv,bv)
   END
; ee_2 ≙
   SELECT  P(gv)
   THEN    GS(gv)
   END
END
```

Within the B approach, an event is considered as the observation of a system transition. Events are spontaneous and show the way a system evolves. An event $e$ is modelled with a guarded substitution: $e \mathrel{\widehat{=}} eG \Longrightarrow eB$ where the predicate $eG$ is a *guard* and the substitution $eB$ is an *action*. It may occur or may be observed only when its guard holds. The guard may use local variables ($bv$) bound to the ANY substitution and global variables ($gv$). The shape of an abstract system is given beside. The semantics of an abstract system lies on its invariant and is guaranteed by proof obligations. The consistency of the system is established by proof obligations:
*i)* the initialization establishes the invariant: $[U]Inv$; *ii)* each event preserves the invariant: $Inv \wedge eG \Longrightarrow [eB]Inv$.

Moreover the events terminate: $I \wedge eG \Rightarrow \mathsf{fis}(eB)$. The predicate $\mathsf{fis}(S)$ expresses that $S$ does not establish *False*: $\mathsf{fis}(S) \Leftrightarrow \neg\,[S]False$. $\mathsf{prd}_v(S)$ is the before-after predicate of the substitution $S$; it relates the values of the state variable just before ($v$) and just after ($v$') the substitution $S$. The deadlock-freeness should be established for an abstract system (the disjunction of the event guards should be true).
The event-based semantics of an abstract system ($A$) is viewed as the event traces of $A$ ($traces(A)$); the set of finite event sequences generated by the evolution of $A$.
As far as practical specification guides are concerned, we introduced in [5, 6] a parallel composition operator to supplement the classical top-down approach of the Event B. This parallel composition allows communication through global variable shared by the abstract (sub)systems. This permits a bottom-up approach for the Event B. This parallel composition is commutative and associative; therefore it is also defined for $n$ systems. More details on the parallel composition can be found in [6].
The B method is supported by theorem provers which are industrial tools (Atelier-B[2] and B-Toolkit[3]). The Event B extension has not dedicated tools but the specifications are translated into classical B and the standard B tools are used.

## 2.2 Overview of the ProB Tool

The ProB tool [11, 12] is an animator and a model checker for B specifications. It provides functionalities to display graphical view of automata. It supports automated consistency checking of B specifications (an abstract machine or a refinement with its state space, its initialization and its operations). The consistency checking is performed on all the reachable states of the machine. The ProB also provides a constraint-based checking; with this approach ProB does not explore the state space from the initializations, it checks whether applying one of the operation can result in an invariant violation independently from the initialization.

The ProB offers many functionalities. The main ones are organized within three categories: *Animation*, *Verification* and *Analysis*. Several functionalities are provided for each category but here, we just list a few of them which are used in this article.

As far as the *Animation* category is concerned, we have the following functionalities:
Random Animation: it starts from an initial state of the abstract machine and then, it selects in a random fashion one of the enabled operations, it computes the next state accordingly and proceeds the animation from this state with one of the enabled operations;

[2] www.clearsy.com
[3] www.b-core.com

View/Reduced Visited States: it displays a minimized graph of the visited states after an animation;
View/Current State: it displays the current state which is obtained after the animation.

In the *Verification* category, the following functionalities are available:
Temporal Model Checking: starting from a set of initialization states (initial nodes), it systematically explores the state space of the current B specification. From a given state (a node), a transition is built for each enabled operation and it ends at a computed state which is a new node or an already existing one. Each state is treated in the same way.
Refi nement Checking: the principle here is based on trace checking. All running traces of the refinement should be traces of the initial specification. Constraint Based Checking: it checks for invariant violation when applying operation independently from initialization states.

As far as the *Analysis* category is concerned, we have the following functionalities:
Compute Coverage: the state space (the nodes) and the transitions of the current specification are checked, some statistics are given on deadlocked states, live states[4], covered and uncovered operations.
Analyse Invariant: it checks if some parts of the current invariant are true or false;
Analyse Properties: the property clause of the current specification is checked.

The ProB tool is used to check liveness properties. Besides, note that if a B prover has been used to perform consistency proof, the invariant should not be violated; the B consistency proof consists in checking that the initialization of an abstract machine establishes the invariant and that all the operations preserve the invariant. In the case where the consistency is not completely achieved ProB can help to discover the faults.

# 3 A Design Support for Multi-Process Systems

A given system may be efficiently specified and verified in B by combining a process-oriented approach and the Event B tools ProB and AtelierB. We introduce a case study to illustrate our proposal.

## 3.1 A Case Study

The system to be studied comes from Milner [13]. The system is made of jobbers and tools (hammer, mallet) and some conveying belts. Two jobbers have to treat the incoming parts (coming on an input belt). The jobbers may use a hammer or a mallet to work the parts. There is only one hammer and only one mallet (they are shared tools/resources). The task of a jobber is as follows: it gets a part, then it gets one of the (free) tool, it deal with the part, it frees the used tool and then the jobber becomes available for another task. The already worked parts leave the system (via an output belt).

We consider an extension of this *Jobber processes* case study: not only two jobbers are considered but several ones. Moreover, new jobbers may enter the system at any time. Now think about the complexity of the specification with a state transition approach. To deal with this complexity, very often state transition approaches make some restrictions on the data and the number of interacting processes.

The formal specification of the case study is not straightforward using Event B. We have to build an abstract system (a state space with a set of events that describe the system evolution).

## 3.2 Practical Limitations of the Separated Approaches

The advantages of each one of the methods involved in the study are underlined above.
**State Transitions.** Capturing a process behaviour is intuitive but state transition systems lack high level structures for complex processes. Handling an undefined, variable number of processes is not tractable. Dealing with several instances of the same processes is not possible. Synchronization of processes should be made explicit.
**B Approach.** A difficult point is that of completeness with respect to event ordering (liveness properties): did the specification covers all the possible evolution (event sequences) expressed in the requirement? Indeed one can have a consistent system (with respect to the stated invariant) which does not meet the desired logical behavioural requirements. Several works investigate the proof of liveness properties of B specifications [8, 9].

[4] those already computed

In the proposed approach, we show that the combination of both approaches help to fight their practical limitations.

## 3.3 The Proposed Approach

The focus is on the correct logical behaviour of the given system; its specification should fulfill the stated informal requirements. Accordingly it should ensure safety and the ordering of the system events should be the one expected. Several steps are distinguished.

**Step 1.** Initialize the construction of an abstract system $A$ that is the formal model of the studied system: a state space $S$ and an event list $E$ are needed. $A$ is a multi-process system; therefore several process types will contribute to define its behaviour.

Identify the system *resources* and distinguish the shared ones. For instance we have a set of parts which are processed: *PART*; a set of tools $TOOL = \{hammer, mallet\}$. The shared resources need a specific access policy. For each kind of these resources, define an event to get/free the resource. Each event has a guard which expresses the conditions and the constraints to get/free the resource ($getHam, getMal, putHam, putMal$).

Identify the set of events that make the system evolves: $\{inPart, outPart, getPart, processPart, getHam, getMal, putHam, putMal, \cdots\}$. These events may be split into classes of events: the general events of the system and those which correspond to the evolution of the identified processes.

**Step 2.** Identify the global properties of the resources with an invariant predicate that characterizes $S$. Complement the invariant cause of $A$.

Identify the properties of the behaviour of the whole system; that is a specific ordering of the events occurrence according to the requirements: *liveness property*.

Let $follow : E \leftrightarrow E$ be a relation that captures the required ordering of the events. It remains to complement *follow* according to the classes of events in $E$.

**Step 3.** Complement $A$ with the B specifications of the general events; for example $inputPart, outputPart$. The first one is specified as follows:

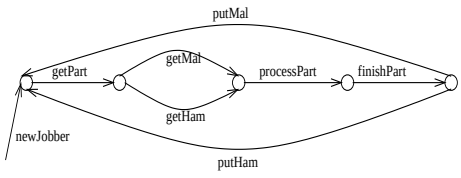
```
inputPart ≙          /* A new part enters the system */
   ANY part WHERE    /* input parts and output parts are disjoints */
      part ∈ PART ∧ part ∉ outParts
   THEN
      inParts := inParts ∪ {part}
   END
```

**Step 4.** Identify the set $\mathcal{P}$ of (types of) processes that interact within $A$. It does not matter the number of the process of each type. For example we have a process type *JOBBER* according to the case study.

**Step 5.** For each process type $P \in \mathcal{P}$ (consider for illustration P as *JOBBER*):

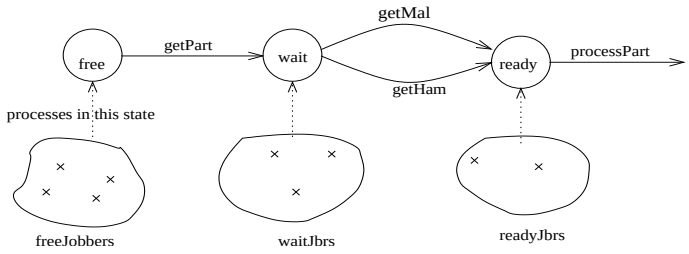
1. use a *process type variable* for the set of processes of this type ($jobbers \subseteq JOBBER$);
2. identify the subset of $E$ events that make the evolution of $P$; for the jobbers we have : $\{getPart, processPart, getHam, getMal, putHam, putMal\}$.
3. define an ordering relation on these events by considering the specific requirements on $P$; this results in describing a subset of *follow* related to $P$. A rather small state transition system may help here. For the *JOBBER* process type we have:

| $follow_{JOBBER}$ | |
|---|---|
| $getPart$ | $\{getHam, getMal\}$ |
| $getHam$ | $\{processPart\}$ |
| $getMal$ | $\{processPart\}$ |
| $processPart$ | $\{finishPart\}$ |
| $finishPart$ | $\{putMal, putHam\}$ |




Some *process state variables* describing disjoint sets of processes are necessary to handle the evolution of the processes.

These variables should be subsets of the previous *process type variable* (with respect to the considered process type). They capture the various states of each process. This ensures (using the event guards) that a process will evolve according to its current state. Several distinct processes may be in the same state; they are recorded in one set. As a process cannot be in different state (see Fig. 3), the variables are disjoints. Therefore, for each process state *ps*, we have a set variable $ps_{processes}$.




As far as the events are concerned, for a given event *ee* which leads (a process) to a state $ps_j$ from a state $ps_i$, the process that provokes this event should be any one process which is in the state $ps_i$. Therefore the specification of the event *ee* has the shape :

**ee** $\widehat{=}$
ANY *pr* WHERE /* one process */
$pr \in ps_{i_{processes}}$
$\wedge \ \cdots$ /* additional predicates */
THEN
$ps_{j_{processes}} := ps_{j_{processes}} \cup \{pr\}$
$\|\ ps_{i_{processes}} := ps_{i_{processes}} - \{pr\}$
END

According to the Jobber case study, we have the following variables.
The variable *freeJobbers* (with $freeJobbers \subseteq jobbers$) denotes the processes which are entirely free; they do not have neither tools nor parts.
The variable *waitJbrs* denotes the jobbers which wait for a tool (they already get a part);
The variables *readyJbrs* describes the set of processes that get a part and that get one tool (hammer or mallet); therefore they are ready to process the part.
The variable *workingj* denotes the processes which are working a part.
Finally *termJbrs* denotes the jobbers which have terminated their task. Note that each one of these variables corresponds to only one state of the process.

4. describe the event on each transition as a B event (guard and substitution). Use the previous (process state) variables to express the guard. For example we have:

**processPart** $\widehat{=}$ /* A part is processed*/
ANY *jbr* WHERE /* one jobber ready to process its part */
$jbr \in jobbers \wedge jbr \in readyJbrs$
$\wedge \ jbr \in \mathrm{dom}(hastool) \wedge jbr \in \mathrm{dom}(hasPart)$
THEN
$workingj := workingj \cup \{jbr\}$
$\|\ readyJbrs := readyJbrs - \{jbr\}$
END

In the same way, the B specifications are described for all the events.

5. complement *A* with the previous set of events. The abstract system *A* is now equipped with the B specifications of all the events.

**Step 6.** Complement and prove the consistency of the *A* abstract system using Atelier B (theorem proving aspect). All the proof obligations should be discharged. However we have not yet the way to guarantee the liveness requirements captured within *follow*. We shall prove that $traces(A)$ coincides with the *follow* relation. That is the role of the following step.
**Step 7.** Analyze and improve the *A* abstract system; this is achieved with the help of animation and model checking with the ProB tool. First, model-check (see 2.2) the *A* abstract system to detect deadlocks. Correct the specification of *A* accordingly. When *A* is deadlock-free, check that it fulfills *follow*. Using the ProB analysis tool (see 2.2) check that all the events are enabled (this corresponds to no uncovered operations). Moreover, each event (*evt*) should enable the events in $follow(evt)$. This is checked by visualizing the reduced visited states (see 2.2). Another way to check this is by stepwise animations; the ProB displays the operations enabled by each activated operation; we should have the correspondence with *follow*. The *A* abstract system is improved accordingly.

As the *follow* relation expresses the set of all possible orderings of the events that could happen in the system, after the **Step 7.**, the logical behaviour analysis is complete and we get a *correct* specification of *A* with respect to the requirements.

**Proof:** The occurrences of event orderings of *A* are checked with respect to *follow*. Formally, the occurrence of an event $e_1 \mathrel{\widehat{=}} eG_1 \Longrightarrow eB_1$ is: $\exists v_i, v_{i+1}.[v := v_i]eG_1 \wedge [v, v' := v_i, v_{i+1}]\mathsf{prd}_v(eB_1)$.

The occurrence of a sequence of two events $e_1.e_2$ is:
$\exists v_i, v_{i+1}, v_{i+2}.[v := v_i]eG_1 \wedge [v, v' := v_i, v_{i+1}]\mathsf{prd}_v(eB_1) \wedge [v := v_{i+1}]eG_2 \wedge [v, v' := v_{i+1}, v_{i+2}]\mathsf{prd}_v(eB_2)$.

This generalizes easily to sequences of *n* events and it is $traces(A)$.
Note that the closure of *follow* (noted $follow^*$) is the event occurrences that correspond to the requirements captured by *follow*. Therefore we have $\boxed{follow^* = traces(A)}$.

In the following we consider further issues about the multi-process specification method.

# 4 Further Issues

In our proposal for multi-process specification, process-oriented approach serves as a basis. Indeed, the latter seems more intuitive with respect to the specification.

## 4.1 Process-oriented Approach

Remind that a *process* is an abstraction of a system behaviour; the latter is described with an *alphabet* and a set of process *operators*. An *alphabet* is the collection of action names. A *communication* is an interaction between several processes to exchange data. There are several operators related to the process-oriented approach.

The widely used operators related to the process-oriented approach are:

- Process definition;
- Inactive process (termination);
- Prefixing;
- Nondeterministic choice (of behaviour);
- Binary parallel composition, n-ary, synchronous or asynchronous communication;
- Process call or recursion;
- Internal action;
- Actions hiding (for synchronisation);
- Conditional;
- Renaming;

- Behaviour interruption;
- Binary or multi-way spawning (like the Unix[5] fork);

Most of the listed operators are already covered by the event B approach (using event specification): inactive process (skip); prefixing (constraining the use of guard); Process call or recursion (using guard).

In the following we consider specifically the nondeterministic choice operator. In previous works we have dealt with the composition of abstract systems [5, 6].

## 4.2 Dealing with Nondeterminism

The expressivity of *follow* is sufficient to tackle non-deterministic aspects.

The *escape* event: for instance an event that may happen at any moment. This situation is simply handled by considering that the *escape* event follows all the events. Therefore it should be present in $follow(e)$ for all event $e$.

The *diamond* events: consider for example that a jobber may get a tool and then get a part or alternatively get a part first and then get a tool. In such situation one event trace excludes the other trace.

The general case is expresses à la process algebra as follows ($\rightarrow$ denotes the sequence, $+$ denotes the choice). Consider for events $e_i$, $e_1$, $e_2$ and $e_f$ such that we have $e_i \rightarrow (e_1 \rightarrow e2 + e_2 \rightarrow e1) \rightarrow e_f$.

That is the event $e_1$ follow $e_2$ unless $e_1$ follows $e_2$. That means we have some constraints to be considered. Consequently, that is to say the guard of the event $e_2$ is not the same according to the evolution traces. Therefore the substitution that will describe the event $e_2$ after one guard is not the same as the substitution that describes the event $e_2$ after the other guard. It is not the same event! They should be distinguished. That is a methodological point.

Consider the situation as sketched in the Fig. 1. One event ($e_2$) has a guard related to the process state $s_i$ and the process reaches the state $s_2$. The other event ($e_2$) has a guard related to the process state $s_1$ and the process reaches the state $s_f$.

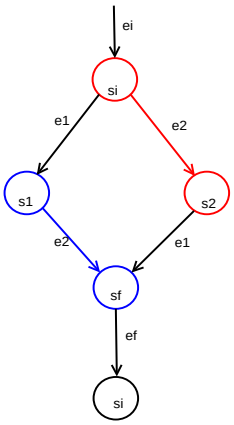



Figure 1: Diamond Events

*The lesson for the specification approach*: the events should be renamed properly so that they are distinct and describe the correct situation.

For instance if we modify the preceding case study by considering that a process jobber can get a tool (*getTool*) and then get a part (*getPart*) or vice versa, the specification will be wrong if we have $follow(getPart) = getTool$ and $follow(getTool) = getPart$.

The solution is to distinguish the events as shown just before; the wright description is obtained with *getPartB4Tool* (i.e get part before the tool), *getPartA3Tool* (i.e. get part after the tool) and similarly *getToolB4Part*, *getToolA3Part*. Therefore we have simply the same description for the *follow* relation (that means without constraining the relation *follow*) :

[5] Unix is a trademark of AT&T.

| *follow* | |
|---|---|
| *getPartB4Tool* | {*getToolA3Part*} |
| *getToolB4Part* | {*getPartA3Tool*} |

As far as process algebra are concerned the same situation is captured through the finite state machine that is computed for a given process. Therefore the evolution of the process is handled with respect to the traces (thus the correct state).

# 5 Summary of Results

We introduce a pragmatic method to guide the specification of multi-process system. Using the described method, we build incrementally a correct B abstract specification of the jobber case study. The feedbacks from model checking help to discover deadlocks and incompleteness of the behaviour obtained with earlier specification versions. The specification is then improved step by step. The B theorem prover is used to discharge all the proof obligations according to the invariant. Liveness properties are expressed using the *follow* relation on events.The ProB tool is used to establish the correctness with respect to liveness. Moreover we handle the dynamic structure of the system as processes may enter and take part of the application at any time. The latter aspect is interesting for studying systems with evolving structure and also for service-oriented systems.

# 6 Conclusion and Perspectives

We presented a practical method as a design support to guide the specification of multi-process systems. The proposal is illustrated with the *jobbers* case study. The obtained specification is proved correct by combining theorem proving via Atelier B and model checking via the ProB tool.

The specificity of the case study, apart of the multi-process aspect, lies on the fact that it describes a generic kind of software systems: several interacting processes that share a set of resources. This is frequently encountered and yet well-known in operating systems: processes accessing the system resources (disks, files, cpu), the dinning philosophers, the readers/writers, etc. These systems are used as the basic blocks for building various software applications including Internet service-oriented ones. The proposed method is yet experimented with several other case studies. We have not consider in this paper refinement of the specification into code, however this step remains in the scope of the standard B approach. But model checking with the ProB tool may also help to faster discharging of refinement proof obligations.

*Perspectives.* The short-term perspective of the work is about fairness properties management. Indeed, the liveness is established but fairness is not guarantee. This property is important in multi-process systems since a starved process may decrease the performance of the system. Think about a situation where a process (*jobber*) has one tool and then is not enabled to work (due to lack of fairness); then the tool it uses is not released to serve other processes. We are investigating the works described in [9, 10] as a basis to complement our study with fairness aspects.

The other short-term perspective is related to the multi-facet analysis beginning from a single abstract reference model. We are studying a method to generate systematically abstract systems from early process-oriented specifications. This method helps for a multi-perspective study of the same initial (B) specification.

# References

[1] J-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] J-R. Abrial. Extending B without Changing it (for developping distributed systems). *Proc. of the 1st Conf. on the B method, H. Habrias (editor), France*, pages 169–190, 1996.

[3] J-R. Abrial. Discrete System Models. Internal Notes (www-lsr.imag.fr/B), February 2002.

[4] J-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *Proc. of the 2nd Conference on the B method, D. Bert (editor)*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer-Verlag, 1998.

[5] C. Attiogbé. A Mechanically Proved Development Combining B Abstract Systems and Spin. In *Proceedings of the 4th International Conference on Quality Software (QSIC 2004)*, pages 42–49. IEEE Computer Society Press, 2004.

[6] C. Attiogbé. A Stepwise Development of the Peterson's Mutual Exclusion Algorithm Using B Abstract Systems. In H. Treharne, S. King, M. Henson, and S. Schneider, eds., *Proc. of ZB'05*, volume 3455 of *LNCS*, pages 124–141. Springer-Verlag, 2005.

[7] R.J. Back and R. Kurki-Suonio. Decentralisation of Process Nets with Centralised Control. In *Proc. of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142. ACM, 1983.

[8] Françoise Bellegarde, Samir Chouali, and Jacques Julliand. Verification of Dynamic Constraints for B Event Systems under Fairness Assumptions. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, eds., *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 477–496. Springer-Verlag, 2002.

[9] H. R. Barradas and D. Bert. Specification and Proof of Liveness Properties under Fairness Assumptions in B Event Systems. In *Proc. of the Integrated Formal Methods (IFM'2002)*, volume 2335 of *LNCS*, pages 360–379, UK, May 2002. Springer-Verlag.

[10] H. R. Barradas and D. Bert. A Fixpoint Semantics of Event Systems with and without Fairness Assumptions. In J.M.T. Romijn, G.P. Smith, and J.C. van de Pol, eds., *Proc. of IFM'2005*, volume 3771 of *LNCS*, pages 327–346, UK, 2005. Springer-Verlag.

[11] M. Leuschel and M. Butler. ProB: A Model Checker for B. In Keijiro A., Stefania G., and Dino M., eds., *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[12] M. Leuschel and E. Turner. Visualizing Larger State Spaces in ProB. In *Proc. of ZB'05*, volume 3455 of *LNCS*, pages 6–23. Springer-Verlag, April 2005.

[13] Robin Milner. *Communication and Concurrency*. Prentice-Hall, NJ, 1989.

# A The ProB Main Interface

The main window (see Fig. 2) of the ProB tool has one text area where the current specification is displayed and three small working area. The area in the middle shows the currently enabled operations. The user may click on the desired operation to proceed with an animation.

# B A First Version of the Abstract System

This version is consistent but not correct with respect to the requirements

```
/*
B Solution of the jobber case (by Milner)
C. Attiogbe - NaBLa Project
December 2005
Version 0 is consistent but incorrect
*/

MACHINE
   jobber0
SETS
```

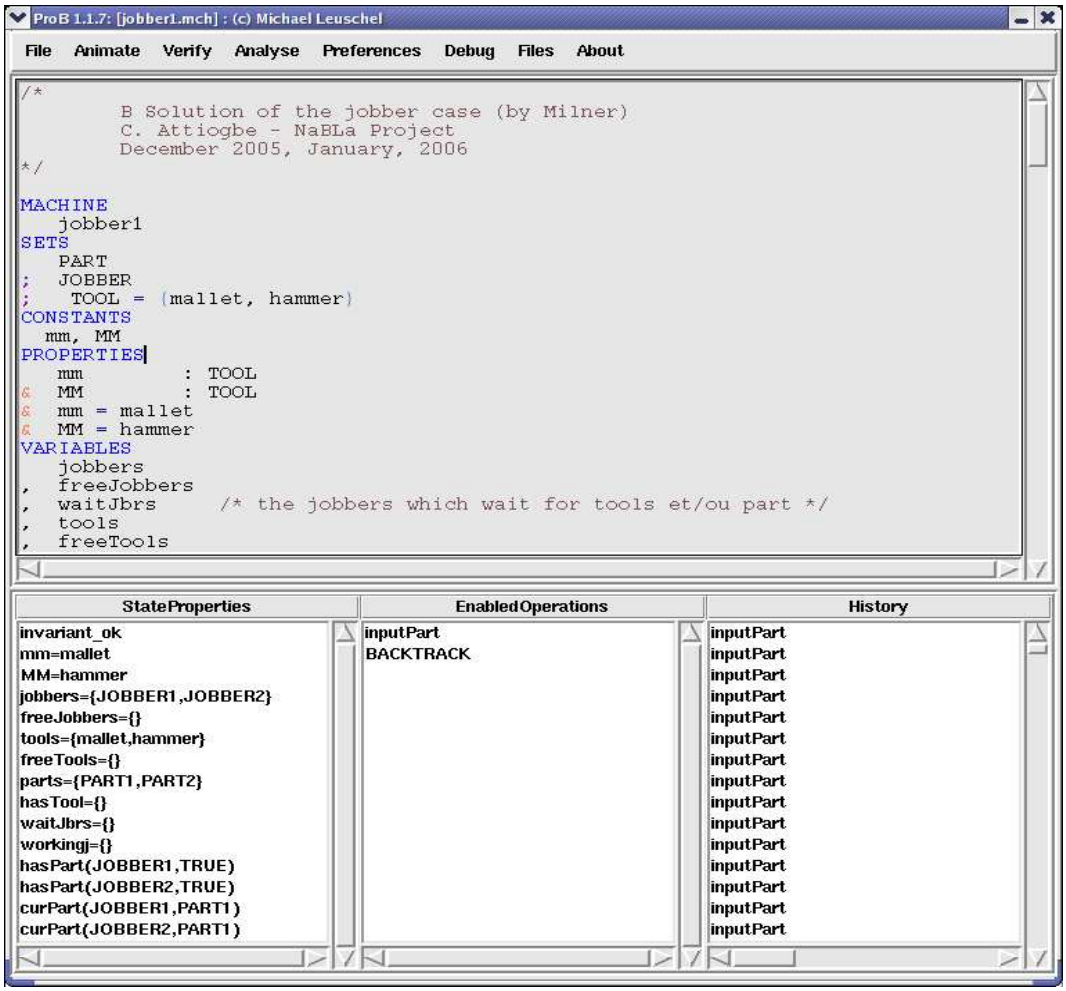



Figure 2: Main ProB Window

```
   PART
;  JOBBER
;   TOOL = {t1, t2}
CONSTANTS
  mm, MM

PROPERTIES
   mm        : TOOL
&  MM        : TOOL
&  mm = t1
&  MM = t2
VARIABLES
   jobbers
,  freeJobbers  /* free jobbers */
,  occJobbers   /* occupied jobbers */
,  waitJbrs     /* the jobbers which wait for tools */
,  tools
,  freeTools
,  parts
,  hasPart     /* jobbers which have part */
,  hasTool
,  curPart
```

```
,  workingj

INVARIANT
   jobbers     <: JOBBER
&  freeJobbers <: JOBBER
&  occJobbers  <: JOBBER
&  freeJobbers <: jobbers
&  occJobbers  <: jobbers
&  freeJobbers /\ occJobbers = {}
&  waitJbrs    <: JOBBER
&  tools       <: TOOL
&  freeTools   <: TOOL
&  freeTools   <: tools
&  parts <: PART  /* the available parts */
&  hasPart : jobbers +-> BOOL
&  hasTool : jobbers +-> TOOL
&  curPart : jobbers +-> PART
&  workingj <: JOBBER    /* the jobbers which are working */


INITIALISATION

 ANY jbrs WHERE
     jbrs <: JOBBER
&    jbrs /= {}
THEN
  jobbers := jbrs
|| freeJobbers := jbrs
|| hasPart := jbrs*{FALSE}
END
||  occJobbers := {}
||  tools     := {mm,MM}
||  freeTools := {mm,MM}
||  parts     := {}
||  hasTool   := {}
||  curPart   := {}
||  waitJbrs  := {}
||  workingj := {}

OPERATIONS

inputPart = /* a part enters the system */
ANY  part
WHERE
part : PART
THEN
parts := parts \/ {part}
END
;
takePart = /* take a part to be processed */
ANY  part, jbr
WHERE  part : parts
&      jbr  : freeJobbers
THEN
parts        := parts -{part}
|| hasPart(jbr) := TRUE
|| curPart(jbr) := part
|| waitJbrs     := waitJbrs \/ {jbr}
END
;
processPart = /* one jobber processes a part */
ANY  jbr
WHERE
jbr : jobbers
& jbr : freeJobbers
```

```
& ((hasTool(jbr) = mm) or (hasTool(jbr) = MM))
&  hasPart(jbr) = TRUE
THEN
workingj       := workingj \/ {jbr}
END
;
getHammer = /* one jobber gets a part */
ANY  jbr, ham
WHERE
jbr : jobbers
& jbr : waitJbrs
& ham : freeTools
& ham = MM
& jbr : waitJbrs
THEN
freeTools    := freeTools - {ham}
|| hasTool(jbr) := ham
|| waitJbrs     := waitJbrs - {jbr}
END
;
        getMallet = /* one jobber gets a part */
ANY  jbr, mal
WHERE
jbr : jobbers
& jbr : waitJbrs
& mal : freeTools
& mal = mm
& jbr : waitJbrs
THEN
freeTools    := freeTools - {mal}
|| hasTool(jbr) := mal
|| waitJbrs     := waitJbrs - {jbr}
END
;
putHammer = /* one jobber gets a part */
ANY  jbr
WHERE
jbr : jobbers
& jbr : workingj
& hasTool(jbr) = MM
THEN
hasTool := hasTool -{ jbr|-> MM}
|| workingj := workingj - {jbr}
END
;
outputPart = /* a (processed) part is output */
ANY jbr, tt
WHERE
jbr : workingj
& tt : tools
& hasTool(jbr) = tt
THEN
freeTools    := freeTools \/ {tt}
|| hasTool := {jbr} <<| hasTool
|| waitJbrs     := waitJbrs \/ {jbr}
END

END
```

# C A Second Version (consistent but incorrect)

This version is also consistent but not correct with respect to the requirements.

The event outputPart is incorrect! but the abstract system is consistent, proved by AtelierB). We find the 'bugs'

using ProB and we correct the abstract system.

For example, we perform an animation for which we have the following coverage (using Analyse/Compute Coverage).

```
live:10
open:9
total:19
TOTAL_OPERATIONS
26
COVERED_OPERATIONS
setup_constants:1
initialise_machine:3
takePart:3
getHammer:1
getMallet:1
processPart:3
putHammer:1
outputPart:1
inputPart:12
UNCOVERED_OPERATIONS
putMallet
```

Note that there is one operation which is not covered : putMallet. This is symptomatic of a possible error. In this case, we check the current state of the system (Animate/View/Current State) and we analyse it (see Fig. 3).

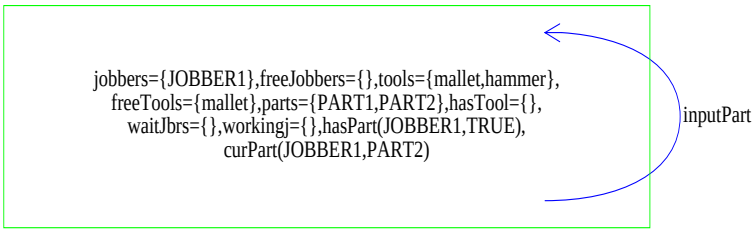



Figure 3: A reached state during animation

```
/*
B Solution of the jobber case (by Milner)
C. Attiogbe - NaBLa Project
December 2005, January, 2006
*/

MACHINE
   jobber1
SETS
   PART
;  JOBBER
;   TOOL = {mallet, hammer}
CONSTANTS
  mm, MM
PROPERTIES
   mm        : TOOL
&  MM        : TOOL
&  mm = mallet
&  MM = hammer
VARIABLES
   jobbers
,  freeJobbers
,  waitJbrs     /* the jobbers which wait for a tool */
,  tools
,  freeTools
```

```
,  parts
,  hasPart
,  hasTool
,  curPart
,  workingj     /* the working jobbers ; have tool and part */

INVARIANT
   jobbers     <: JOBBER
&  freeJobbers <: JOBBER
&  workingj <: JOBBER    /* the jobbers which are working */
&  freeJobbers <: jobbers
&  waitJbrs    <: JOBBER
&  workingj  <: jobbers
&  freeJobbers /\  workingj = {}
&  freeJobbers /\  waitJbrs = {}
&  tools       <: TOOL
&  freeTools   <: TOOL
&  freeTools   <: tools
&  parts <: PART  /* the available parts */
&  hasPart : jobbers +-> BOOL
&  hasTool : jobbers +-> TOOL
&  curPart : jobbers +-> PART
&  mm /= MM

INITIALISATION

ANY jbrs WHERE
jbrs <: JOBBER
THEN
jobbers := jbrs
||  freeJobbers := jbrs
||  hasPart := jbrs*{FALSE}
END
||  tools     := {mm,MM}
||  freeTools := {mm,MM}
||  parts     := {}
||  hasTool   := {}
||  curPart   := {}
||  waitJbrs  := {}
||  workingj  := {}

OPERATIONS

inputPart =  /* input of a part to be treated */
ANY  part
WHERE
part : PART
THEN
parts := parts \/ {part}
END
;
takePart = /* A free jobber (without tool) takes a part to be processed */
ANY  part, jbr
WHERE  part : parts
&      jbr  : freeJobbers
THEN
parts        := parts -{part}
|| hasPart(jbr) := TRUE
|| curPart(jbr) := part
|| waitJbrs     := waitJbrs \/ {jbr}
||  freeJobbers  := freeJobbers - {jbr}
END
;
processPart = /* A jobber (with a tool and a part) processes a part */
ANY  jbr
```

```
WHERE
jbr : jobbers
& jbr /: freeJobbers
& jbr : dom(hasTool) /* ((hasTool(jbr) = mm) or (hasTool(jbr) = MM))*/
&  hasPart(jbr) = TRUE
THEN
workingj       := workingj \/ {jbr}
END
;
getHammer = /* A jobber (with a part) gets the hammer */
ANY  jbr, ham
WHERE
jbr : jobbers
& jbr : waitJbrs   /* ie it has a part */
& ham : freeTools
& ham = MM
& jbr /: dom(hasTool)
THEN
freeTools    := freeTools - {ham}
|| hasTool(jbr) := ham
||      waitJbrs     := waitJbrs - {jbr}
/* jbr is ready, not waiting */
END
;
        getMallet =  /* A jobber gets the mallet */
ANY  jbr, mal
WHERE
jbr : jobbers
& jbr : waitJbrs  /* ie it has a part */
& mal : freeTools
& mal = mm
& jbr /: dom(hasTool)
THEN
freeTools    := freeTools - {mal}
|| hasTool(jbr) := mal
|| waitJbrs := waitJbrs - {jbr}
/* now jbr is ready, */
END
;
putHammer = /* free the hammer, when the job is terminated */
ANY  jbr
WHERE
jbr : jobbers
& jbr : workingj
& hasTool(jbr) = MM
THEN
hasTool := hasTool - {jbr|-> MM}
|| workingj := workingj - {jbr}
END
;
putMallet = /* free the mallet, avant de finir?? */
ANY  jbr
WHERE
jbr : jobbers
& jbr : workingj
& hasTool(jbr) = mm
THEN
hasTool := hasTool - {jbr|-> mm}
|| workingj := workingj - {jbr}
END
;
outputPart = /* output of a part already treated, this free the jobber/tool */
ANY jbr, tt
WHERE
jbr : workingj
```

```
/* may finish jbr */
& jbr /: waitJbrs
& tt : tools
& hasTool(jbr) = tt
THEN
freeTools    := freeTools \/ {tt}
||       workingj := workingj - {jbr}
|| freeJobbers := freeJobbers \/ {jbr}
|| hasTool := {jbr} <<| hasTool
END

END
```

The following (see Fig. 4) is a (reduced) graph from an animation.

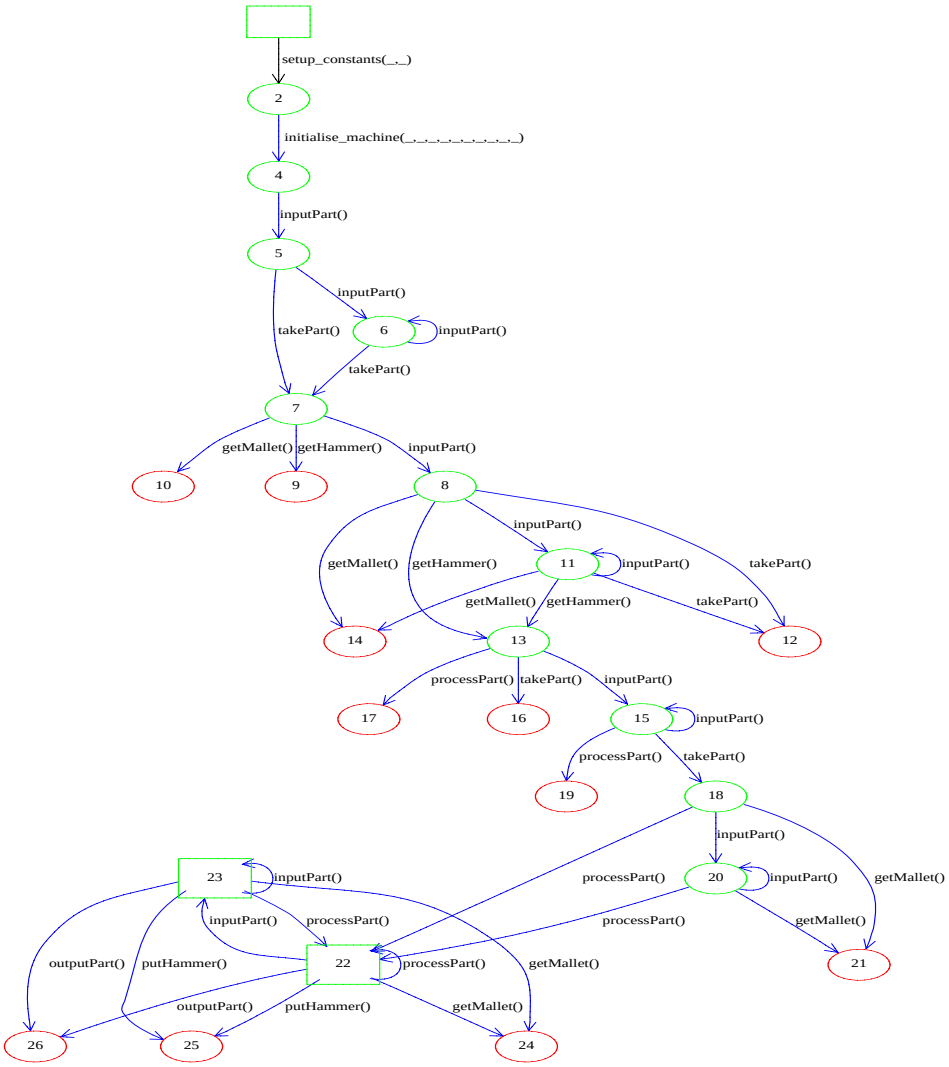



Figure 4: A graph from an animation

# D The Correct Version

```
/*
B Solution of the jobber case (by Milner)
C. Attiogbe - NaBLa Project
*/

MACHINE
   jobber
SETS
   PART
;  JOBBER
;   TOOL = {mallet, hammer}
CONSTANTS
  mm, MM
PROPERTIES
   mm        : TOOL
&  MM        : TOOL
&  mm = mallet
&  MM = hammer
VARIABLES
   jobbers
,  freeJbrs
,  waitJbrs     /* the jobbers which wait for tools et/ou part */
,  readyJbrs    /* the jobbers which are ready for processing */
,  termJbrs     /* the jobbers which have terminated their jobs*/
,  tools
,  freeTools
,  inParts
,  outParts
,  hasPart
,  hasTool
,  curPart
,  workingj     /* the working jobbers ; have tool and part */

INVARIANT
   jobbers     <: JOBBER
&  freeJbrs <: JOBBER
&  workingj <: JOBBER    /* the jobbers which are working */
&  readyJbrs <: JOBBER
&  termJbrs <: JOBBER
&  freeJbrs <: jobbers
&  waitJbrs <: JOBBER
&  workingj <: jobbers
&  freeJbrs /\  workingj = {}
&  freeJbrs /\  waitJbrs = {}
&  tools       <: TOOL
&  freeTools   <: TOOL
&  freeTools   <: tools
&  inParts <: PART  /* the available parts */
&  outParts <: PART  /* the available parts */
&  inParts /\ outParts = {}
&  hasPart : jobbers +-> BOOL
&  hasTool : jobbers +-> TOOL
&  curPart : jobbers +-> PART
&  mm /= MM

INITIALISATION

ANY jbrs WHERE
jbrs <: JOBBER
THEN
jobbers := jbrs
||  freeJbrs := jbrs
||  hasPart := jbrs*{FALSE}
```

```
END
||  tools     := {mm,MM}
||  freeTools := {mm,MM}
||  inParts   := {}
||  outParts  := {}
||  hasTool   := {}
||  curPart   := {}
||  waitJbrs  := {}
||  workingj  := {}
||  readyJbrs := {}
||  termJbrs  := {}

OPERATIONS

inputPart =  /* input of a part to be treated */
ANY  part
WHERE
part : PART & part /: outParts
THEN
inParts := inParts \/ {part}
END
;
getPart = /* A free jobber (without tool) takes a part to be processed */
ANY  part, jbr
WHERE  part : inParts
&      jbr  : freeJbrs
THEN
inParts      := inParts -{part}
|| hasPart(jbr) := TRUE
|| curPart(jbr) := part
|| waitJbrs     := waitJbrs \/ {jbr}
||  freeJbrs     := freeJbrs - {jbr}
END
;
getHammer = /* A jobber (with a part) gets the hammer */
ANY  jbr, ham
WHERE
jbr : jobbers
& jbr : waitJbrs   /* ie it has a part */
& ham : freeTools
& ham = MM
& jbr /: dom(hasTool)
THEN
freeTools    := freeTools - {ham}
|| hasTool(jbr) := ham
||       waitJbrs     := waitJbrs - {jbr}
|| readyJbrs    := readyJbrs \/  {jbr}
END
;
        getMallet =  /* A jobber gets the mallet */
ANY  jbr, mal
WHERE
jbr : jobbers
& jbr : waitJbrs  /* ie it has a part */
& mal : freeTools
& mal = mm
& jbr /: dom(hasTool)
THEN
freeTools    := freeTools - {mal}
|| hasTool(jbr) := mal
|| waitJbrs     := waitJbrs - {jbr}
|| readyJbrs    := readyJbrs \/  {jbr}
END
;
processPart = /* A jobber (with a tool and a part) processes a part */
```

```
ANY  jbr
WHERE
jbr : jobbers
& jbr : readyJbrs
&    jbr /: freeJbrs
& jbr : dom(hasTool) /* ie ((hasTool(jbr) = mm) or (hasTool(jbr) = MM)) */
&  hasPart(jbr) = TRUE
THEN
workingj  := workingj \/ {jbr}
||  readyJbrs := readyJbrs -{jbr}
/* || may finish(jbr) */
END
;
finishPart = /* A jobber */
ANY  jbr,cp
WHERE
jbr : workingj
& cp : PART
& cp = curPart(jbr)
& cp /: inParts
THEN
termJbrs := termJbrs \/ {jbr}
|| workingj := workingj - {jbr}
|| outParts := outParts \/ {cp}
END
;
putHammer = /* free the hammer, when the job is terminated */
ANY  jbr
WHERE
jbr : jobbers
&   jbr : termJbrs
& jbr /: workingj
& jbr /: waitJbrs
& hasTool(jbr) = MM
THEN
hasTool := hasTool - {jbr|-> MM}
|| termJbrs := termJbrs - {jbr}
|| freeTools :=  freeTools \/ {MM}
|| tools     := tools \/ {MM}
|| freeJbrs  := freeJbrs \/ {jbr}
||  curPart := {jbr} <<| curPart
END
;
putMallet = /* free the mallet, avant de finir?? */
ANY  jbr
WHERE
jbr : jobbers
&   jbr : termJbrs
& jbr /: workingj
& jbr /: waitJbrs
& hasTool(jbr) = mm
THEN
hasTool := hasTool - {jbr|-> mm}
|| termJbrs := termJbrs - {jbr}
|| freeTools := freeTools \/ {mm}
|| tools     := tools \/ {mm}
|| freeJbrs  := freeJbrs \/ {jbr}
||  curPart := {jbr} <<| curPart
END
;
outputPart = /* output of an already treated part, this frees the jobber/tool */
ANY  tp
WHERE
tp : outParts
THEN
```

```
outParts := outParts -{tp}
END
;
inJobber =
ANY  jbr
WHERE
jbr : JOBBER
& jbr /: (jobbers \/freeJbrs)
&  jbr /: waitJbrs
THEN
jobbers := jobbers \/ {jbr}
|| freeJbrs := freeJbrs \/ {jbr}
END
END
```

# Combining B Tools for Multi-Process Systems Specification

**Christian Attiogbé**

**Abstract**

We introduce a systematic method that combines process-oriented design and abstract systems to specify multi-process system using Event B. As far as specification is concerned, the proposed method guides the user to describe his/her system by considering both process-oriented view and event B style of specification.
With regard to mechanization, the method enforces the conjoined use of both theorem proving technique and model checking technique with the associated tools: AtelierB and ProB. This method makes it easy the specification in Event B of multi-process systems and it also fasters the correctness proof. Our proposal is illustrated with a case study: a multi-process system with the interaction between processes that deal with parts using common resources.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms: Multi-Process Specification, Event B, Theorem Proving, Model checking

Additional Key Words and Phrases: Multi-Process Specification, Event B, Theorem Proving, Model checking

LINA, Université de Nantes
2, rue de la Houssinière
B.P. 92208 — F-44322 NANTES CEDEX 3